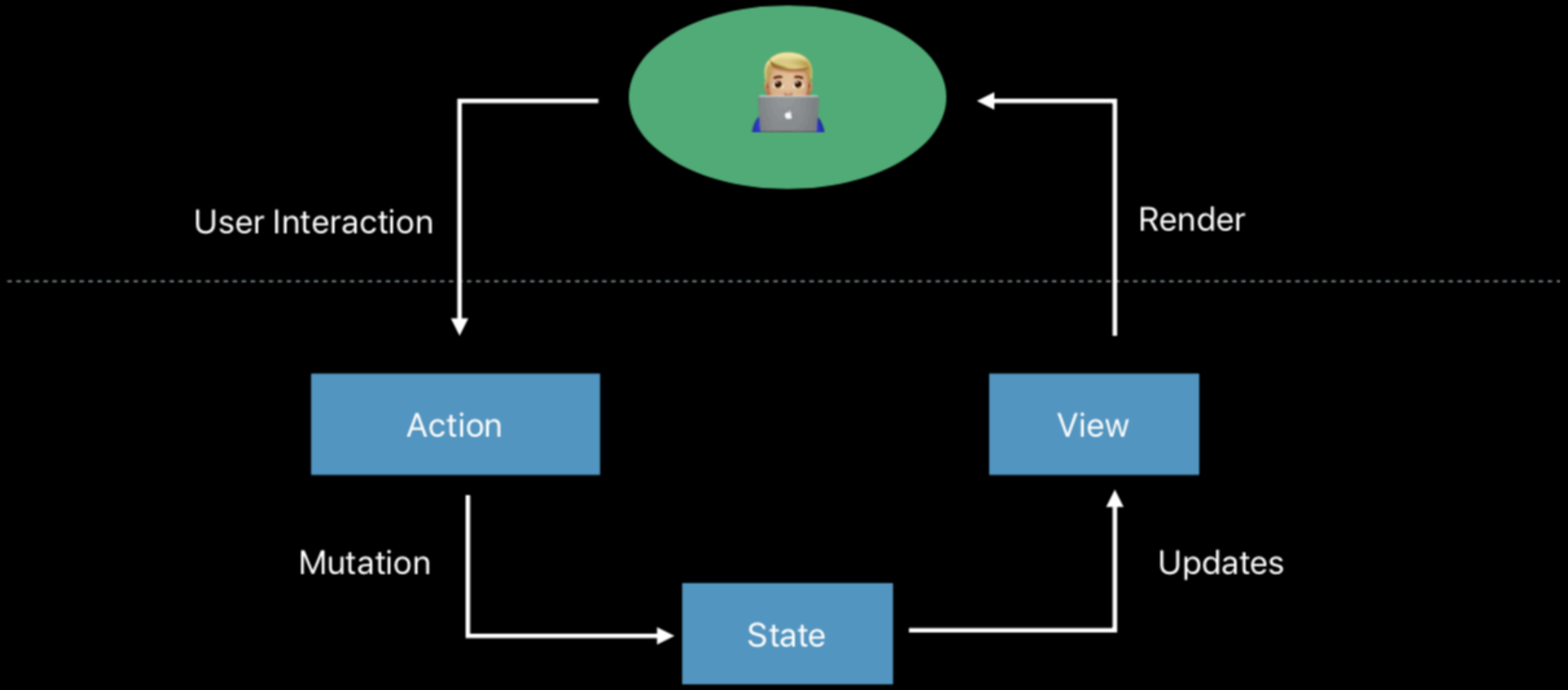
Unidirectional Data Flow Through SwiftUI

@kitasuke


User Interaction
Render
Action
View
Mutation
State
Updates

SwiftUI ≈ React

~~SwiftUI ≈ React~~

SwiftUI ≈ React + MobX

MobX

Simple, scalable state management

Events invoke actions. Actions are the only thing that modify state and may have other side effects.

```
@action onClick = () => {
  this.props.todo.done = true;
}
```

State is observable and minimally defined. Should not contain redundant or derivable data. Can be a graph, contain classes, arrays, refs, etc.

```
@observable todos = [{
  title: "learn MobX",
  done: false
}]
```

Computed values are values that can be derived from the state using a pure function. Will be updated automatically by MobX and optimized away if not in use.

```
@computed get completedTodos() {
  return this.todos.filter(
    todo => todo.done
  )
}
```

Reactions are like computed values and react to state changes. But they produce a side effect instead of a value, like updating the UI.

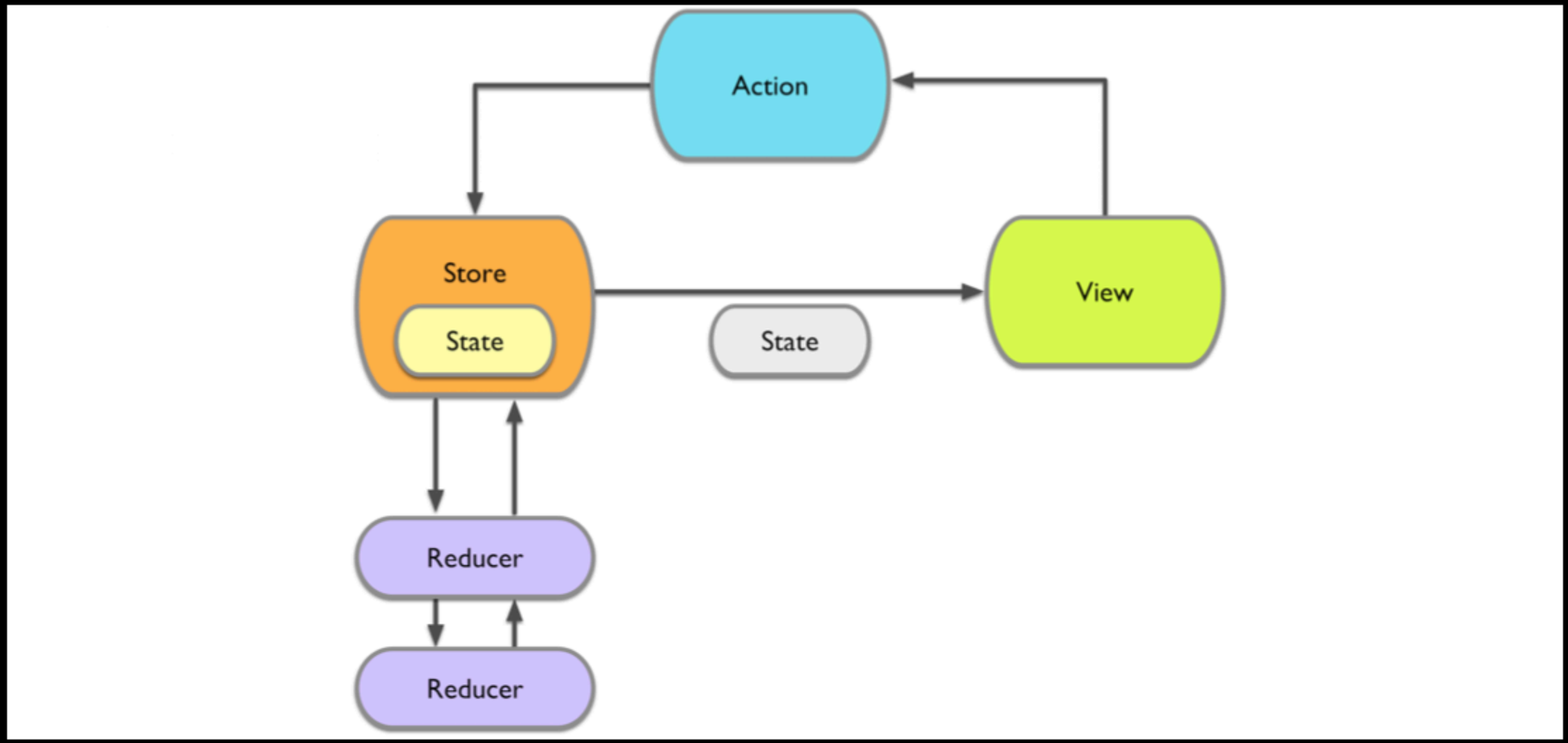
```
const Todos = observer({ todos } =>
  <ul>
    todos.map(todo => <TodoView ... />
  </ul>
)
```

Redux

A predictable state container

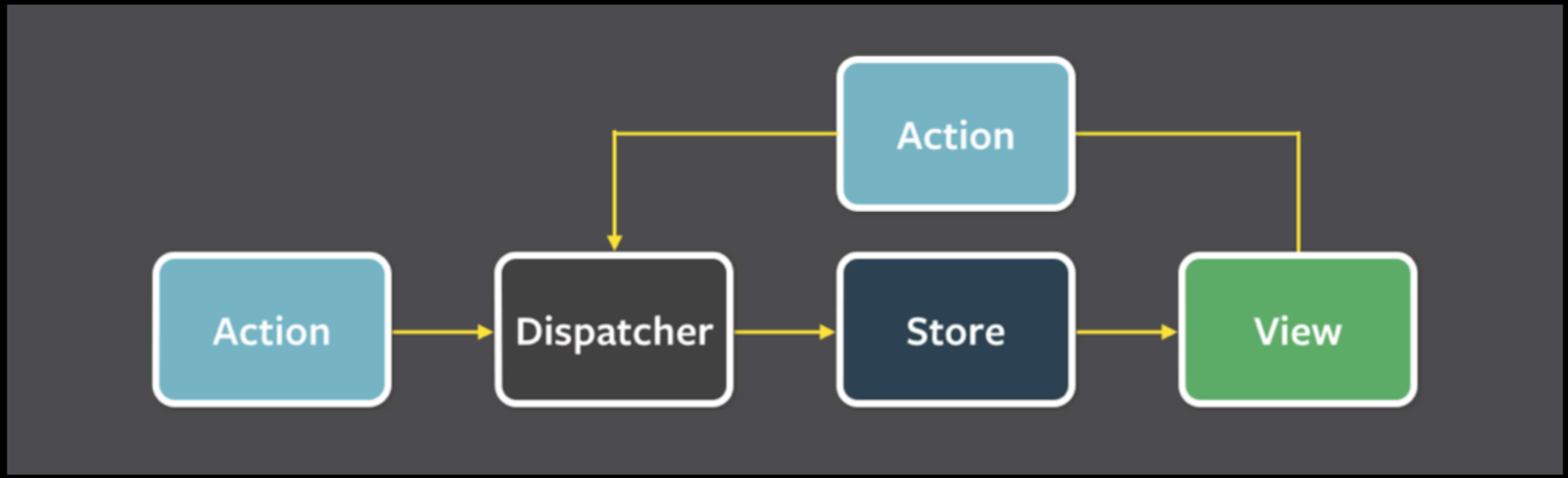
ReSwift

Unidirectional Data Flow in Swift - Inspired by Redux

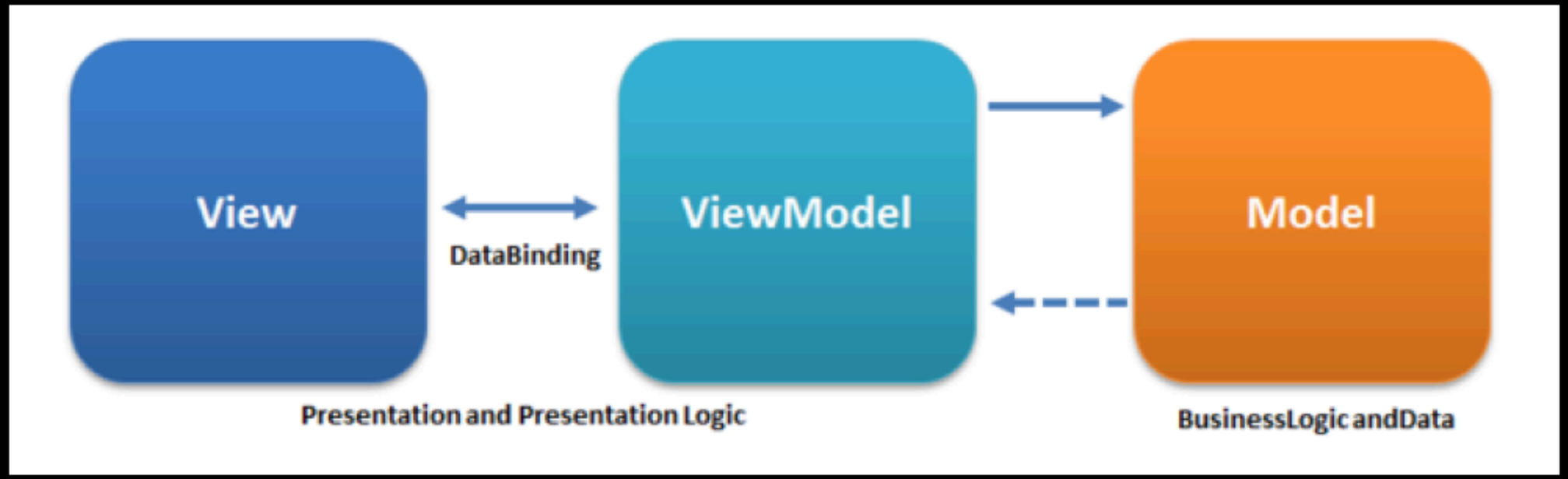

Action
Store
State
State
View
Reducer
Reducer

Flux

Application architecture for building user interfaces


Action
Action
Dispatcher
Store
View

MVVM


View
DataBinding
ViewModel
Model
Presentation and Presentation Logic
BusinessLogic andData

Why do we need these design patterns?

Unidirectional data flow

Unidirectional data flow

Unidirectional data flow

→ Decoupled

Unidirectional data flow

→ Decoupled

→ Testable

Unidirectional data flow

→ Decoupled

→ Testable

→ Scalable

What patterns can we adopt with SwiftUI?

SwiftUI + Flux

View

```
struct RepositoryListView : View {
    @ObjectBinding var store: RepositoryListStore = .shared
    private var actionCreator: RepositoryListActionCreator

    var body: some View {
        NavigationView {
            List(store.repositories) { repository in
                RepositoryListRow(repository: repository)
            }
        }
        .onAppear(perform: { self.actionCreator.onAppear() })
    }
}
```

ActionCreator

```swift
final class RepositoryListActionCreator {
    private let dispatcher: RepositoryListDispatcher
    private let onAppearSubject = PassthroughSubject<Void, Never>()

    init(dispatcher: RepositoryListDispatcher = .shared) {
        _ = onAppearSubject
            .map { ... }
            .sink(receiveValue: { [dispatcher] in
                // dispatch action
                dispatcher.dispatch(.updateRepositories($0))
            })
    }

    func onAppear() {
        onAppearSubject.send(())
    }
}
```

ActionCreator

```
final class RepositoryListActionCreator {
    private let dispatcher: RepositoryListDispatcher
    private let onAppearSubject = PassthroughSubject<Void, Never>()

    init(dispatcher: RepositoryListDispatcher = .shared) {
        _ = onAppearSubject
            .map { ... }
            .sink(receiveValue: { [dispatcher] in
                // dispatch action
                dispatcher.dispatch(.updateRepositories($0))
            })
    }

    func onAppear() {
        onAppearSubject.send(())
    }
}
```

ActionCreator

```swift
final class RepositoryListActionCreator {
    private let dispatcher: RepositoryListDispatcher
    private let onAppearSubject = PassthroughSubject<Void, Never>()

    init(dispatcher: RepositoryListDispatcher = .shared) {
        _ = onAppearSubject
            .map { ... }
            .sink(receiveValue: { [dispatcher] in
                // dispatch action
                dispatcher.dispatch(.updateRepositories($0))
            })
    }

    func onAppear() {
        onAppearSubject.send(())
    }
}
```

Action

```
enum RepositoryListAction {
    case updateRepositories([Repository])
}
```

Dispatcher

```
final class RepositoryListDispatcher {
    static let shared = RepositoryListDispatcher()
    private let actionSubject = PassthroughSubject<RepositoryListAction, Never>()

    func register(callback: @escaping (RepositoryListAction) -> ()) {
        _ = actionSubject.sink(receiveValue: callback)
    }

    func dispatch(_ action: RepositoryListAction) {
        actionSubject.send(action)
    }
}
```

Dispatcher

```
final class RepositoryListDispatcher {
    static let shared = RepositoryListDispatcher()
    private let actionSubject = PassthroughSubject<RepositoryListAction, Never>()

    func register(callback: @escaping (RepositoryListAction) -> ()) {
        _ = actionSubject.sink(receiveValue: callback)
    }

    func dispatch(_ action: RepositoryListAction) {
        actionSubject.send(action)
    }
}
```

Dispatcher

```
final class RepositoryListDispatcher {
    static let shared = RepositoryListDispatcher()
    private let actionSubject = PassthroughSubject<RepositoryListAction, Never>()

    func register(callback: @escaping (RepositoryListAction) -> ()) {
        _ = actionSubject.sink(receiveValue: callback)
    }

    func dispatch(_ action: RepositoryListAction) {
        actionSubject.send(action)
    }
}
```

Store

```
final class RepositoryListStore: BindableObject {
    static let shared = RepositoryListStore()
    let didChange = PassthroughSubject<Void, Never>()

    private(set) var repositories: [Repository] = [] {
        didSet { didChange.send(()) }
    }

    init(dispatcher: RepositoryListDispatcher = .shared) {
        dispatcher.register { [weak self] action in
            switch action {
            case .updateRepositories(let repositories):
                self?.repositories = repositories
            }
        }
    }
}
```

Store

```
final class RepositoryListStore: BindableObject {
    static let shared = RepositoryListStore()
    let didChange = PassthroughSubject<Void, Never>()

    private(set) var repositories: [Repository] = [] {
        didSet { didChange.send(()) }
    }

    init(dispatcher: RepositoryListDispatcher = .shared) {
        dispatcher.register { [weak self] action in
            switch action {
            case .updateRepositories(let repositories):
                self?.repositories = repositories
            }
        }
    }
}
```

Store

```
final class RepositoryListStore: BindableObject {
    static let shared = RepositoryListStore()
    let didChange = PassthroughSubject<Void, Never>()

    private(set) var repositories: [Repository] = [] {
        didSet { didChange.send(()) }
    }

    init(dispatcher: RepositoryListDispatcher = .shared) {
        dispatcher.register { [weak self] action in
            switch action {
            case .updateRepositories(let repositories):
                self?.repositories = repositories
            }
        }
    }
}
```

View

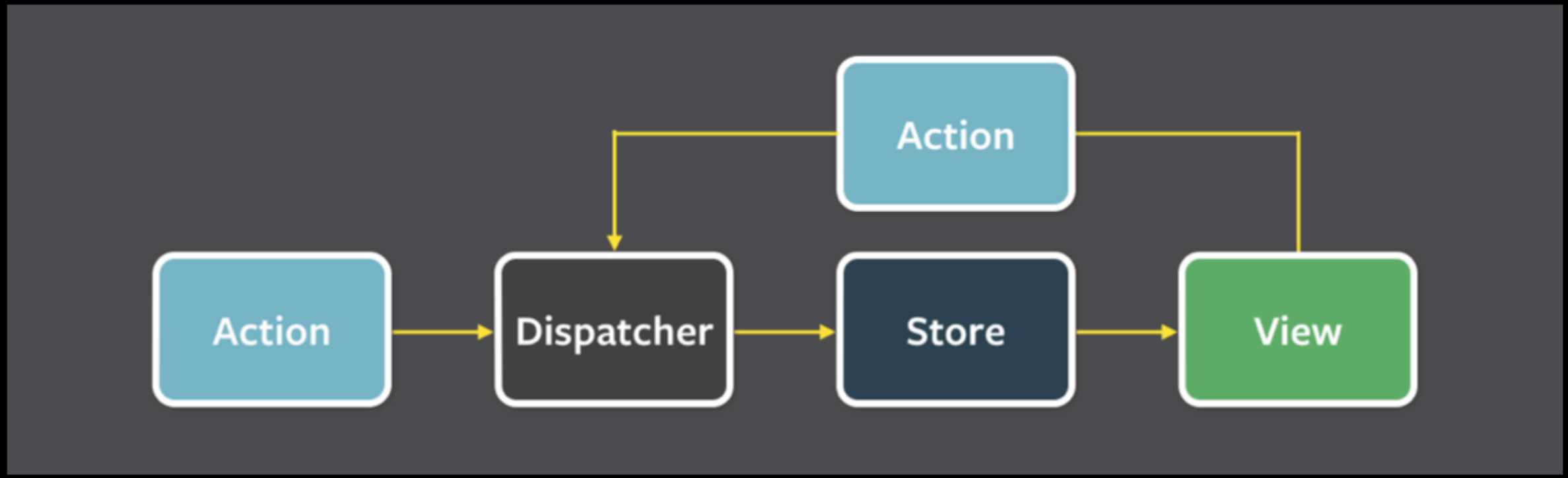
```
struct RepositoryListView : View {
    @ObjectBinding var store: RepositoryListStore = .shared
    private var actionCreator: RepositoryListActionCreator

    var body: some View {
        NavigationView {
            List(store.repositories) { repository in
                RepositoryListRow(repository: repository)
            }
        }
        .onAppear(perform: { self.actionCreator.onAppear() })
    }
}
```

View

```
struct RepositoryListView : View {
    @ObjectBinding var store: RepositoryListStore = .shared
    private var actionCreator: RepositoryListActionCreator

    var body: some View {
        NavigationView {
            List(store.repositories) { repository in
                RepositoryListRow(repository: repository)
            }
        }
        .onAppear(perform: { self.actionCreator.onAppear() })
    }
}
```


Action
Action
Dispatcher
Store
View

SwiftUI + MVVM

View

```
struct RepositoryListView : View {
    @ObjectBinding var viewModel: RepositoryListViewModel

    var body: some View {
        NavigationView {
            List(viewModel.output.repositories) { repository in
                RepositoryListRow(repository: repository)
            }
        }
        .onAppear(perform: { self.viewModel.apply(.onAppear) })
    }
}
```

UnidirectionalDataFlowType

```
protocol UnidirectionalDataFlowType {
    associatedtype InputType
    associatedtype OutputType

    func apply(_ input: InputType)
    var output: OutputType { get }
}
```

UnidirectionalDataFlowType

```
protocol UnidirectionalDataFlowType {
    associatedtype InputType
    associatedtype OutputType

    func apply(_ input: InputType)
    var output: OutputType { get }
}
```

UnidirectionalDataFlowType

```
protocol UnidirectionalDataFlowType {
    associatedtype InputType
    associatedtype OutputType

    func apply(_ input: InputType)
    var output: OutputType { get }
}
```

ViewModel

```
final class RepositoryListViewModel:
    BindableObject, UnidirectionalDataFlowType {
    typealias InputType = Input

    enum Input {
        case onAppear
    }
    func apply(_ input: Input) {
        switch input {
        case .onAppear: onAppearSubject.send(())
        }
    }
    private let onAppearSubject = PassthroughSubject<Void, Never>()
    ...
}
```

ViewModel

```
final class RepositoryListViewModel:
    BindableObject, UnidirectionalDataFlowType {
    typealias OutputType = Output

    struct Output {
        var repositories: [Repository] = []
    }
    private(set) var output = Output() {
        didSet {
            didChangeSubject.send(())
        }
    }
    ...
}
```

ViewModel

```
final class RepositoryListViewModel:
    BindableObject, UnidirectionalDataFlowType {

    init() {
        let repositoriesStream = onAppearSubject
            .map { ... }
            .assign(to: \.output.repositories, on: self)
        ...
    }
}
```

View

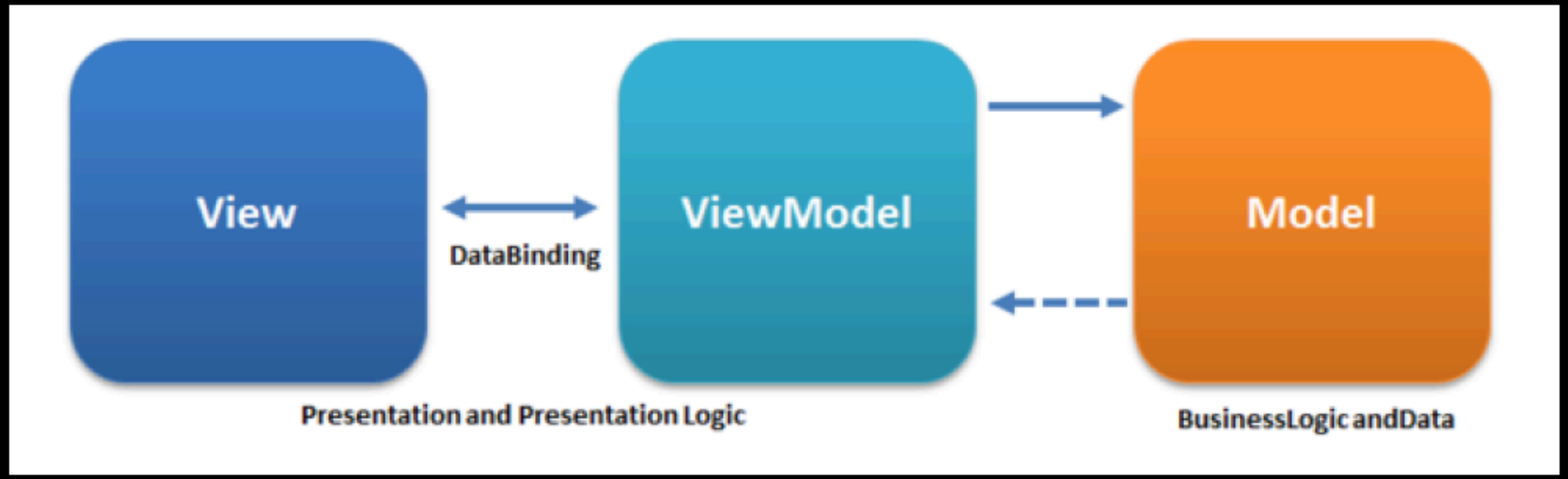
```
struct RepositoryListView : View {
    @ObjectBinding var viewModel: RepositoryListViewModel

    var body: some View {
        NavigationView {
            List(viewModel.output.repositories) { repository in
                RepositoryListRow(repository: repository)
            }
        }
        .onAppear(perform: { self.viewModel.apply(.onAppear) })
    }
}
```

View

```
struct RepositoryListView : View {
    @ObjectBinding var viewModel: RepositoryListViewModel

    var body: some View {
        NavigationView {
            List(viewModel.output.repositories) { repository in
                RepositoryListRow(repository: repository)
            }
        }
        .onAppear(perform: { self.viewModel.apply(.onAppear) })
    }
}
```


View
DataBinding
ViewModel
Model
Presentation and Presentation Logic
BusinessLogic andData

SwiftUI + Redux

View

```
struct RepositoryListView : View {
    @ObjectBinding var state: RepositoryListState
    let reduxStore: ReduxStore

    var body: some View {
        NavigationView {
            List(state.repositories) { repository in
                RepositoryListRow(repository: repository)
            }
        }
        .onAppear(perform: {
            self.reduxStore.dispatch(RepositoryListAction.requestAsyncCreator())
        })
    }
}
```

Action

```
enum RepositoryListAction: Action {
    case updateRepositories([Repository])

    static func requestAsyncCreator() -> RequestActionCreator {
        return { (_, store: DispatchingStoreType) in
            return ThunkAction(
                Future<Action, Never> { promise in
                    _ = APIService().searchRepository()
                        .map { RepositoryListAction.updateRepositories($0) }
                        .sink(receiveValue: { promise(.success($0)) })
                }
            )
        }
    }
}
```

Reducer

```
struct RepositoryListReducer {
    static func reduce(action: Action,
        state: RepositoryListState) -> RepositoryListState {
        switch action {
        case let action as RepositoryListAction:
            switch action {
            case .updateRepositories(let repositories):
                state.repositories = repositories
            }
        default: break
        }
        return state
    }
}
```

State

```
final class RepositoryListState: StateType, BindableObject {
    let didChange = PassthroughSubject<Void, Never>()

    var repositories: [Repository] = [] {
        didSet { didChange.send(()) }
    }
}
```

View

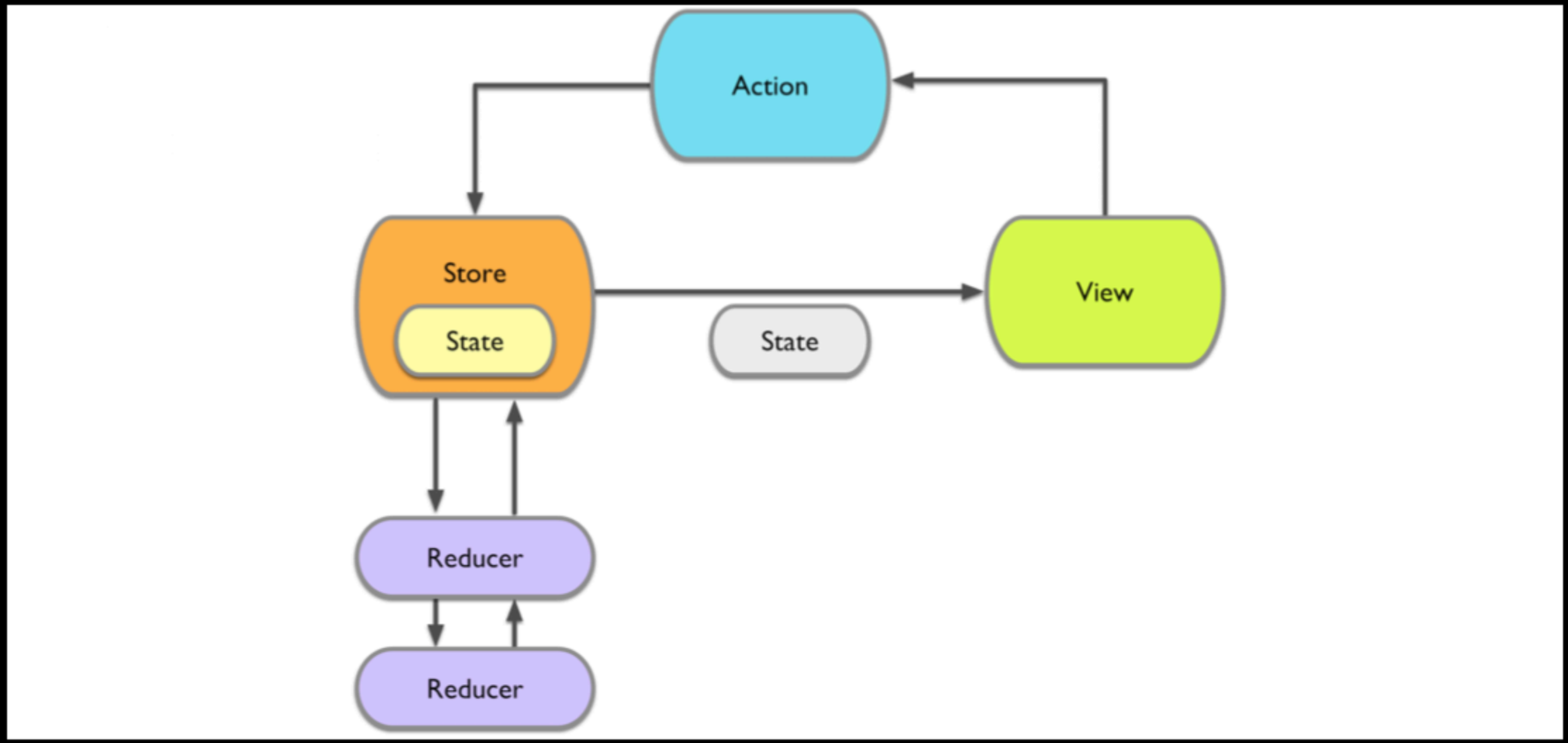
```
struct RepositoryListView : View {
    @ObjectBinding var state: RepositoryListState
    let reduxStore: ReduxStore

    var body: some View {
        NavigationView {
            List(state.repositories) { repository in
                RepositoryListRow(repository: repository)
            }
        }
        .onAppear(perform: {
            self.reduxStore.dispatch(RepositoryListAction.requestAsyncCreator())
        })
    }
}
```

View

```
struct RepositoryListView : View {
    @ObjectBinding var state: RepositoryListState
    let reduxStore: ReduxStore

    var body: some View {
        NavigationView {
            List(state.repositories) { repository in
                RepositoryListRow(repository: repository)
            }
        }
        .onAppear(perform: {
            self.reduxStore.dispatch(RepositoryListAction.requestAsyncCreator())
        })
    }
}
```


Action
Store
State
State
View
Reducer
Reducer

See more details

→ kitasuke/SwiftUI-Flux

→ kitasuke/SwiftUI-MVVM

→ kitasuke/SwiftUI-Redux

Takeaways

- → Unidirectional data flow is simple, but robust
  - → SwiftUI provides nice APIs for data binding
- → These design patterns are suitable for SwiftUI

References

→ https://developer.apple.com/xcode/swiftui/

→ https://mobx.js.org

→ https://redux.js.org

→ https://github.com/ReSwift/ReSwift

→ https://facebook.github.io/flux/